Delegating Responsibility in Digital Systems: Horton's "Who Done It?"

Mark S. Miller
Hewlett-Packard Labs

Jed Donnelley
NERSC, LBNL

Alan H. Karp
Hewlett-Packard Labs

May 10, 2007

Programs do good things, but also do bad,
making software security more than a fad.
The authority of programs, we do need to tame.
But bad things still happen. Who do we blame?

From the very beginnings of access control,
should we be safe by construction?
Or should we patrol?
Horton shows how, in an elegant way,
we can simply do both, and so save the day.

1 Introduction

There are two approaches to protect users from the harm programs can cause, *proactive control* and *reactive control* [22]. Proactive control is intended to prevent bad things from happening, or to limit the damage when it does. But when repeated patterns of abuse occur, we need some workable notion of "who" to blame, so we can reactively suspend the access of the responsible party. For example, granting read-only access to a wiki proactively prevents the recipient from modifying the contents. Knowing "who" posted spam allows reactively suspending that party's write access.

In the 1960's and 1970's the dominant access control paradigms were capbilities and Access Control Lists (ACLs). A capability—like an object-reference in a memory-safe language—is an unforgeable token used both to designate some object and to provide access to that object. Because the term "capabilities" has since been used for many alternative access control rules [10], we now refer to the original pure model [1] as *object-capabilities* or *ocaps* for short.

ACL systems consider a program to be acting on behalf of its "user". Access is allowed or disallowed by checking whether this operation on this resource is permitted for this user.

By allowing the controlled delegation of authority, ocap systems support proactive control well. The invoker of an object normally passes as arguments just those objects that the receiver needs to carry out that request. A user can likewise delegate to an application just that portion of the user's authority the application needs [20], limiting damage should it be corrupted by a virus. But because ocaps operate on an anonymous "bearer right" basis; they seem to make reactive control impossible. Indeed, although many of the historical criticisms of ocaps have since been refuted [10, 9, 15], a remaining unrefuted criticism is that they cannot record who to blame for which action [5].

Only ACLs currently support reactive control. By tagging all actions with the identity of the user they are supposedly serving, they can log who to blame, and whose access to suspend. But ACL systems are weak at proactive control. Solitaire runs with all its user's privileges. If it runs amok, it could do its user great harm.

The lack of reactive control has been an important enough problem for people to forego the advantages of ocaps. One answer would be to try to mix elements of the two paradigms in one security architecture. There have been many such attempts [6, 3]; perhaps some day one of these will bear fruit. Another is to emulate some of the virtues of one paradigm as a pattern built on the other. For example, Polaris [18] uses lessons learned from ocaps to limit the authority of ACL-based applications for Windows, as Plash [14] does for Unix, without modifying either these applications or their underlying ACL OS.

In this paper, we show how to attribute actions in an ocap system. As a tribute to Dr. Seuss [4], we call our protocol Horton (*H*igher-*O*rder *R*esponsibility *T*racking of *O*bjects in *N*etworks). Horton can be interposed between existing ocap-based application objects, without modifying either these objects or their underlying ocap foundations. Horton supports the tracking of responsibility and the reactive suspension of access based on attributed abuse. Horton thereby refutes this criticism of the ocap paradigm. Horton also makes the full chain of responsibility available, something ACL systems do not do.

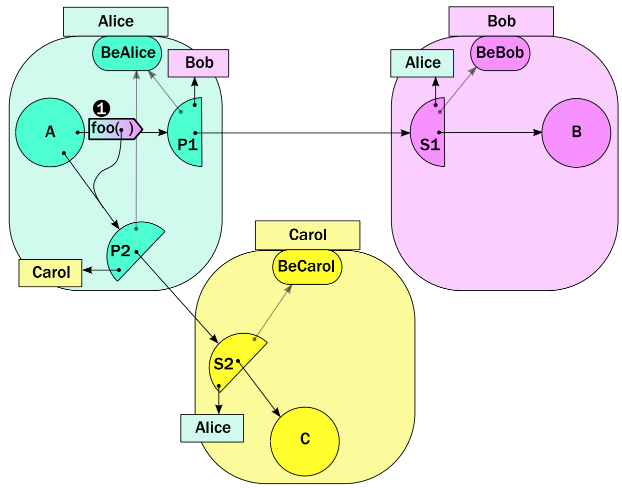
2 The Horton Protocol

Every protocol which builds some sort of secure relationship between their players must face two issues. The base case, building an initial secure relationship between players not yet connected by this protocol, and the inductive case, in which a new secure relationship is bootstrapped from earlier assumed-secure relationships. Horton contributes nothing to the issues of initial connectivity, so this paper only treats the inductive case.

As with object computation, ocap references are conveyed as arguments in messages from a sender to a receiver. Here, we examine a scenario in which a sender, object A in step ❶, executes `b.foo(c)`, "thinking" it is sending the message "foo" to receiver B with a reference to object C as an argument.

Our round objects, A, B, and C, are application-level objects unaware of Horton. The boxes represent messages in flight. Other shapes depict parts of the Horton infrastructure. When a proxy (an outgoing half circle such as P1) receives an app-level message, it encodes and sends it (Figure 2, ❹) in a `deliver` message to its corresponding stub (an incoming half circle such as S1). The stub decodes it into an app-level message which it delivers to its target object (Figure 3, ❻). For Horton to be transparent, the message delivered to B in step ❻ must have the same app-level meaning as the message sent by A in step ❶. To complete the induction, the relationship represented by the new B→P3→S3→C path must have the security we need, *assuming* that the A→P1→S1→B and A→P2→S2→C paths already have this security.

To support reactive security, we need to attribute actions to responsible identities. Cryptographic protocols often represent an electronic identity as a key pair. For example, a public encryption key identifies whoever knows the corresponding private decryption




```
def makeProxy(beMe, whoBlame, stub) {
  def proxy implements Proxy {
    to getGuts() {                        # as P2
      # beMe=BeAlice whoBlame=Carol stub=S2
12:   return [whoBlame, stub]}
❶:  match [verb, [p2 :Proxy]] {          # as P1
      # beMe=BeAlice whoBlame=Bob stub=S1
11:   def [carolWho, s2] := p2.getGuts()
❷:    def gs3 := s2.intro(whoBlame)
32:   def p3Desc := ["t", gs3, carolWho]
      # ...log request to Bob...
❹:    stub.deliver(verb, [p3Desc])}}
  return proxy}
```

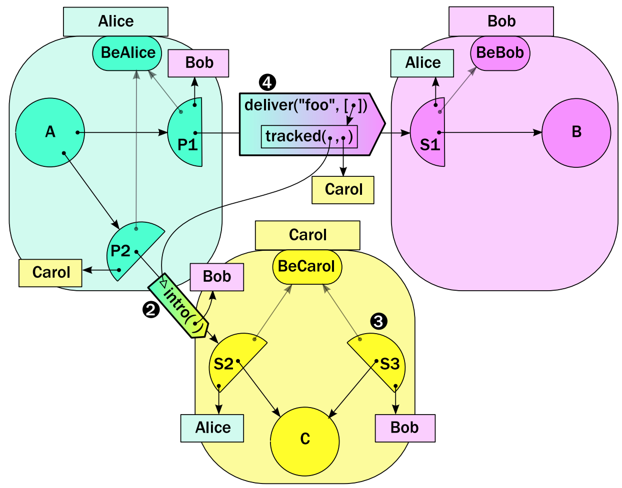
key. Put another way, knowledge of a decryption key provides the ability to *be* (or *speak for* [8]) the entity identified by the corresponding encryption key. In ocap systems, the sealer/unsealer pattern [11] provides a similar logic. Rectangles such as the one labeled “Alice” represent Who objects, providing a `seal(contents)` operation, returning an opaque *box* encapsulating the contents. The corresponding BeAlice object provides the authority to *be* the entity identified by Alice’s Who object. BeAlice provides an `unseal(box)` operation that returns the contents of a box sealed by Alice’s Who. The large round rectangles and colors aggregate all objects whose behavior should be blamed on a given Who.

A complete Horton implementation in Java is available from `erights.org/download/horton/`. For expository purposes, this paper uses E to show just the Horton code needed for the illustrated case: sending a proxy as the single argument of a message with no return result. The line numbers on the code show the sequence of steps taken by our example. A diagram step ×10 = (the corresponding line number), i.e., (❷)+1 = (21). Mostly, this simplified code uses just the simple sequential five-minute subset of E explained in [9, Ch6: A Taste of E]. We also use a some reflection which we explain as needed.

We need reflection immediately. When the `foo` message arrives at proxy P1, it does not match any of the method definitions, so it falls through to the `match` clause (❶), which receives messages reflectively. The clause’s head is a pattern matched against a pair of the message name (here, `"foo"`) and the list of arguments (here, a list holding only proxy P2).

P1 asks P2 for the value of P2’s `whoBlame` and `stub` fields, which hold Carol’s Who and S2 (11,12). P1 then sends `intro(bobWho)` to S2 (❷), by which Alice is saying in effect *“Carol, I’d like to share with Bob my access to C. Could you create a stub for Bob’s use?”* Nothing forces Alice to share her rights in this indirect way; Alice *could* just give Bob direct access to S2. But then Carol would *necessarily* blame Alice for Bob’s use of S2, which Alice might not like.

Carol makes S3 for Bob’s use of C (21). Carol tags S3 with Bob’s Who, so Carol can blame Bob for messages sent to S3. Carol then “gift wraps” S3 for Bob and returns the gift-wrapped S3 (`gs3`) to Alice




```
def makeStub(beMe, whoBlame, targ) {
  def stub {
❷: to intro(bobWho) {                  # as S2
      # beMe=BeCarol whoBlame=Alice targ=C
      # ...log Alice delegating to Bob...
21:   def s3 := makeStub(beMe,bobWho,targ)
❸:    return wrap(s3, bobWho, beMe)}
❹: to deliver(verb, [p3Desc]) {        # as S1
      # beMe=BeBob whoBlame=Alice targ=B
      # ...log access by Alice...
41:   def [=="t", gs3, carolWho] := p3Desc
❺:    def s3 := unwrap(gs3, carolWho, beMe)
59:   def p3 := makeProxy(beMe,carolWho,s3)
❻:    E.call(targ, verb, [p3])}}
  return stub}
```

as the result of the `intro` message (❸). Alice includes `gs3` in the `p3Desc` record encoding the p2 argument of the original message (32). By including this in the `deliver` request to Bob’s S1 (❹), Alice is saying in effect *“Bob, please unwrap this to get the ability to use an object provided by Carol.”*

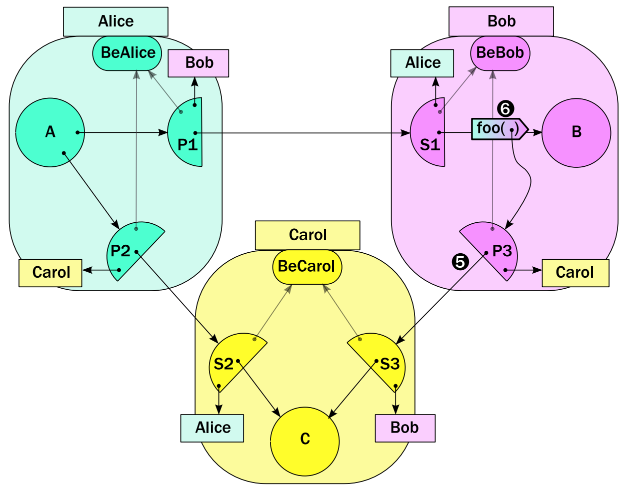
Bob’s S1 unpacks the record (41), unwraps `gs3` to get S3 (❺), which it uses to make proxy P3 (59). Bob tags P3 with Carol’s Who, so Bob can blame Carol for the behavior of S3. S1 then includes P3 as the argument of the app-level `foo` message sent to B using E’s reflective `E.call` primitive (❻).

Clearly, the `unwrap` function must be the inverse of the `wrap` function. Using identity functions would be simplest, but would also give Alice access to S3. This would enable Alice to fool Carol into blaming Bob for messages Alice sends to S3. If Carol also believes she has independent evidence that Bob is not a pseudonym of Alice's, then this mis-attribution would be a loss of security. (Without such evidence this mis-attribution would not matter, since Carol would blame Alice for Bob's behavior anyway.)

Carol should at least gift-wrap S3 so only Bob can unwrap it. Could we simply use the `seal`/`unseal` operations of Bob's who/be pair as the `wrap`/`unwrap` functions? Unfortunately, this would still enable Alice to give Bob a gift allegedly from Carol, but which Bob unwraps to obtain a faux S3 created by Alice.

In our solution, Carol's `wrap` creates a `provide` function, seals it so only Bob can unseal it, and returns the resulting box as the wrapped gift (31). Bob's `unwrap` unseals it to get a `provide` function allegedly from Carol (51). Bob will need to call `provide` (54) so that *only* Carol can provide S3 to him. Bob declares an `s3hole` variable (52), and a `fill` function for Carol to call to fill in this hole with S3. He seals this in a box only Carol can unseal (53) and passes this to `provide` (54). Carol's `provide` unseals it to get Bob's `fill` function (55), which Carol can call (56) to fill in the `s3hole` with S3 (57). After Carol's `provide` returns, Bob's `unwrap` returns whatever it finds in the `s3hole` (58).

Should Bob and Carol ever come to know each other independently of Alice, they can then blame each other, rather than Alice, for actions logged by P3 and S3. Say C is a wiki page. If Carol decides that Bob has spammed this page, Carol could then revoke Bob's access without revoking Alice's access by shutting off S3. If Bob decides that C is flaky, he can stop using Carol's services by shutting off proxies such as P3. This completes the induction.




```
❸: def wrap(s3, bobWho, beCarol) {  # as S2
      def provide(fillBox) {
55:     def fill := beCarol.unseal(fillBox)
56:     fill(s3)}
31:   return bobWho.seal(provide)}
❺: def unwrap(gs3,carolWho,beBob){  # as S1
51:   def provide := beBob.unseal(gs3)
52:   var s3hole := null
57:   def fill(s3) {s3hole := s3}
53:   def fillBox := carolWho.seal(fill)
54:   provide(fillBox)
58:   return s3hole}
```

3 Related Work

Some distributed ocap systems interpose objects to serialize/deserialize messages [2, 13], stretching the reference graphic between local ocap systems. Secure Network Objects [19] and Client Utility [7] leveraged their intermediation to add some identity tracking. Horton unbundles such policy-based intermediation as a separately composable abstraction.

Reactive security ocap patterns include the logging forwarder [16] and the caretaker [12]. These patterns, however, provide no means for Alice to ask Carol to issue a separately accountable access to Bob.

Petmail [21] and SPKI [3] also provide similar features as Horton in a non-ocap environment. They also show how petnames [17] can enable secure human interpretation of the identities. Future Horton extensions should similarly support petnames.

4 Conclusions

Delegation is a fundamental part of human society. If digital systems are to mediate ever more of our interactions, we must be able to delegate responsibility within them. While some systems support the controlled delegation of authority, and other systems support assignment of responsibility, today we have no means for delegating responsibility, that is, delegating authority coupled with assigning responsibility for using that authority. Horton demonstrates how delegation of responsibility can be added to systems that already support delegation of authority—object-capability systems.

5 Acknowledgments

We thank the e-lang and cap-talk communities, especially Tyler Close, Norm Hardy, Chip Morningstar, Jonathan Shapiro, Terry Stanley, Marc Stiegler, Bill Tulloh, Brian Warner, Meng Weng Wong, and Ka-Ping Yee.

References

[1] J. B. Dennis and E. C. V. Horn. Programming Semantics for Multiprogrammed Computations. Technical Report MIT/LCS/TR-23, M.I.T. Laboratory for Computer Science, 1965.

[2] J. E. Donnelley. A Distributed Capability Computing System. In *Proc. Third International Conference on Computer Communication*, pages 432–440, Toronto, Canada, 1976.

[3] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI Certificate Theory (IETF RFC 2693), Sept. 1999.

[4] T. S. Geisel. *Horton Hears a Who!* Random House Books for Young Readers, 1954.

[5] V. D. Gligor, J. C. Huskamp, S. Welke, C. Linn, and W. Mayfield. Traditional capability-based systems: An analysis of their ability to meet the trusted computer security evaluation criteria. Technical report, National Computer Security Center, Institute for Defense Analysis, 1987.

[6] P. A. Karger and A. J. Herbert. An Augmented Capability Architecture to Support Lattice Security and Traceability of Access. In *Proc. 1984 IEEE Symposium on Security and Privacy*, pages 2–12, Oakland, CA, Apr. 1984. IEEE.

[7] A. H. Karp, R. Gupta, G. Rozas, and A. Banerji. The Client Utility Architecture: The Precursor to E-Speak. Technical Report HPL-2001-136, Hewlett Packard Laboratories, June 09 2001.

[8] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in Distributed Systems: Theory and Practice. *ACM Trans. Comput. Syst.*, 10(4):265–310, 1992.

[9] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[10] M. S. Miller, K.-P. Yee, and J. S. Shapiro. Capability Myths Demolished. Technical Report Report SRL2003-02, Systems Research Laboratory, Department of Computer Science, Johns Hopkins University, mar 2003.

[11] J. H. Morris, Jr. Protection in Programming Languages. *Communications of the ACM*, 16(1):15–21, 1973.

[12] D. D. Redell. *Naming and Protection in Extensible Operating Systems*. PhD thesis, Department of Computer Science, University of California at Berkeley, Nov. 1974.

[13] R. D. Sansom, D. P. Julin, and R. F. Rashid. Extending a Capability Based System into a Network Environment. In *Proc. 1986 ACM SIGCOMM Conference*, pages 265–274, Aug. 1986.

[14] M. Seaborn. Plash: The Principle of Least Authority Shell, 2005. `plash.beasts.org/`.

[15] A. Spiessens. *Patterns of Safe Collaboration*. PhD thesis, Université catholique de Louvain, Louvain la Neuve, Belgium, February 2007.

[16] M. Stiegler. A picturebook of secure cooperation, 2004. `www.skyhunter.com/marcs/SecurityPictureBook.ppt`.

[17] M. Stiegler. An Introduction to Petname Systems. In *Advances in Financial Cryptography Volume 2*. Ian Grigg, 2005.

[18] M. Stiegler, A. H. Karp, K.-P. Yee, T. Close, and M. S. Miller. Polaris: virus-safe computing for windows xp. *Commun. ACM*, 49(9):83–88, 2006.

[19] L. van Doorn, M. Abadi, M. Burrows, and E. P. Wobber. Secure Network Objects. In *Proc. 1996 IEEE Symposium on Security and Privacy*, pages 211–221, 1996.

[20] D. Wagner and E. D. Tribble. A Security Analysis of the Combex DarpaBrowser Architecture, Mar. 2002. `www.combex.com/papers/darpa-review/`.

[21] B. Warner. Petmail. *Codecon*, 2004. `petmail.lothar.com/design.html`.

[22] K.-P. Yee. Firefighters and engineers. *interactions*, 13(3):48–49, 2006.